UNIVERSITY AT ALBANY
State University of New York

COLLEGE OF ENGINEERING AND APPLIED SCIENCES
DEPARTMENT OF COMPUTER SCIENCE
ICSI213/IECE213 Data Structures

Project 03 Created by Qi Wang

Click here for the project discussion recording.

Table of Contents

- **Part I: General Project Information**
  All projects are individual projects unless it is notified otherwise.
- All projects must be submitted via Blackboard. No late projects or e-mail submissions or hard copies will be accepted.
- Two submission attempts will be allowed on Blackboard. **Only the last attempt will be graded.**
- Work will be rejected with no credit if
  - The project is late.
  - The project is not submitted properly (wrong files, not in required format, etc.). For example,
    - The submitted file can't be opened.
    - The submitted work is empty or wrong work.
    - Other issues.
  - The project is a copy or partial copy of others' work (such as work from another person or the Internet).
- **Students must turn in their original work. Any cheating violation will be reported to the college. Students can help others by sharing ideas, but not by allowing others to copy their work.**
- Documents to be submitted as a zipped file:
  - **UML class diagram(s)** – created with Violet UML or StarUML
  - **Java source file(s) with Javadoc style inline comments**
  - **Supporting files if any** (For example, files containing all testing data.)
- Students are required to submit a design, all error-free source files with Javadoc style inline comments, and supporting files. Lack of any of the required items will result in a really low credit or no credit.
- Your TA will grade, and then post the feedback and the grade on Blackboard if you have submitted it properly and on time. If you have any questions regarding the feedback or the grade, please reach out to the TA first. You may also contact the instructor for this matter.

Part II: Project grading rubric

Components	**Max points**
UML Design (See an example in part II.)	Max. 10 points
Javadoc Inline comments (See an example in part II.)	Max. 10 points
The rest of the project	Max. 40 points

All projects will be evaluated based upon the following software development activities.

Analysis:

- Does the software meet the exact specification / customer requirements?
- Does the software solve the exact problem?

Design:

- Is the design efficient?

Code:

- Are there errors?
- Are code conventions followed?
- Does the software use the minimum computer resource (computer memory and processing time)?
- Is the software reusable?
- Are comments completely written in Javadoc format?
    a. Class comments must be included in Javadoc format before a class header.
    b. Method comments must be included in Javadoc format before a method header.
    c. More inline comments must be included in either single line format or block format inside each method body.
    d. All comments must be completed in correct format such as tags, indentation etc.

Debug/Testing:

- Are there bugs in the software?

Documentation:

- Complete all documentations that are required.

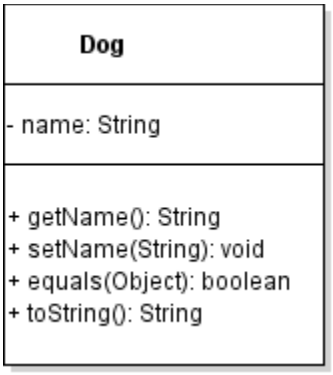
Part III: Examples on how to complete a project from start to finish

To complete a project, the following steps of a software development cycle should be followed. These steps are not pure linear but overlapped.

Analysis-design-code-test/debug-documentation.

1) Read project description to understand all specifications(**Analysis**).
2) Create a design (an algorithm for method or a UML class diagram for a class) (**Design**)
3) Create Java programs that are translations of the design. (**Code/Implementation**)
4) Test and debug, and (**test/debug)**
5) Complete all required documentation. (**Documentation**)

The following shows a sample design. The corresponding source codes with inline Javadoc comments are included on next page. How to test/debug a software is included on the following pages.


Dog
- name: String
+ getName(): String
+ setName(String): void
+ equals(Object): boolean
+ toString(): String

```
import java.util.Random;

/**
 * Representing a dog with a name.
 * @author Qi Wang
 * @version 1.0
 */
public class Dog{
    /**
     * The name of this dog
     */
    private String name;


    /**
     * Constructs a newly created Dog object that represents a dog with an empty name.
     */
    public Dog(){
        this("");
    }
    /**
     * Constructs a newly created Dog object wit
     * @param name The name of this dog
     */
    public Dog(String name){
        this.name = name;
    }

    /**
     * Returns the name of this dog.
     * @return The name of this dog
     */
    public String getName(){
        return this.name;
    }

    /**
     * Changes the name of this dog.
     * @param name The name of this dog
     */
    public void setName(String name){
        this.name = name;
    }

    /**
     * Returns a string representation of this dog. The returned string contains the type of
     * this dog and the name of this dog.
     * @return A string representation of this dog
     */
    public String toString(){
        return this.getClass().getSimpleName() + ": " + this.name;
    }


    /**
     * Indicates if this dog is "equal to" some other object. If the other object is a dog,
     * this dog is equal to the other dog if they have the same names. If the other object is
     * not a dog, this dog is not equal to the other object.
     * @param obj A reference to some other object
     * @return A boolean value specifying if this dog is equal to some other object
     */
    public boolean equals(Object obj){
        //The specific object isn't a dog.
        if(!(obj instanceof Dog)){
            return false;
```

Class comments must be written in Javadoc format before the class header. A **description** of the class, author information and version information are required.

Comments for fields are required.

open {

TAB

TAB

Method comments must be written in Javadoc format before the method header. the first word must be a **capitalized** verb in the **third** person. Use punctuation marks properly.

open {

TAB

TAB

A description of the method, comments on parameters if any, and comments on the return type if any are required.

A Javadoc comment for a formal parameter consists of three parts:

- parameter tag,
- a name of the formal parameter in the design ,
  (The name must be consistent in the comments and the header.)
- and a phrase explaining what this parameter specifies.

A Javadoc comment for return type consists of two parts:

- return tag,
- and a phrase explaining what this returned value specifies

More inline comments can be included in single line or block comments format in a method.

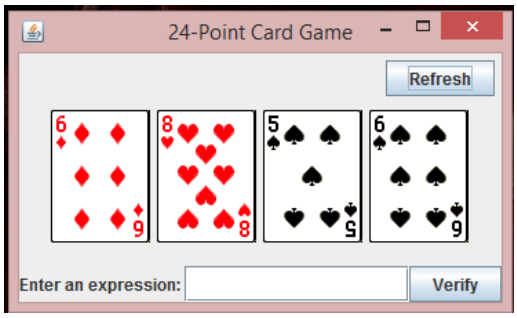
```
            }

            //The specific object is a dog.
            Dog other = (Dog)obj;
            return this.name.equalsIgnoreCase(other.name);
        }
}
```

Part IV: Project description

Project 3 The 24-point card game

The 24-point card game is to pick any 4 cards from 52 cards, as shown in the figure below. Note that the Jokers are excluded. Each card represents a number. An Ace, King, Queen, and Jack represent **1**, **13**, **12**, and **11**, respectively. You can click the *Refresh* button to get four new cards.



Specification:

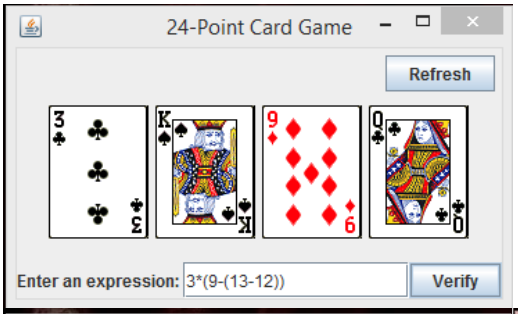
Enter an expression that uses the four numbers from the four selected cards. Each number must be used once and only once. You can use the operators (addition, subtraction, multiplication, and division) and parentheses in the expression. The expression must evaluate to 24. After entering the expression, click *Verify* button to check whether the numbers in the expression is correct. Display the verification in a message window (see figures below).

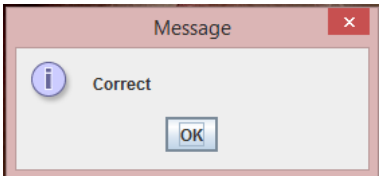
Assume that images are stored in files named 1.png, 2.png, … , 52.png, in the order of spades, hearts, diamonds, and clubs. So, the first 13 images are for spades 1, 2, 3… 13.

Assume that no spaces are entered before the first token, in between tokens, and after the last token.

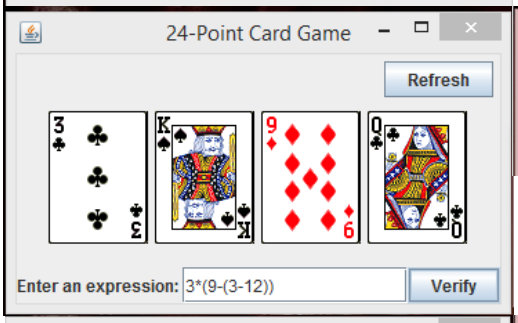
Assume that only numbers can be entered as operands.

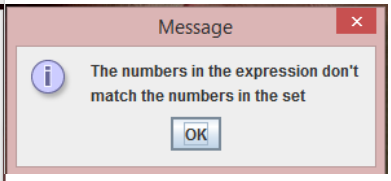
Numbers match and the result is 24.



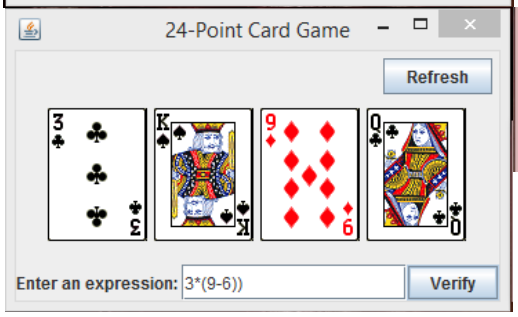


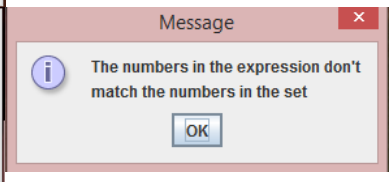
Numbers don't match.





Numbers don't match.





More example,

If 2,3,4,5 are selected and displayed, user can try one of the following expressions to win.

2*((3+4)+5)

2*(3+(4+5))

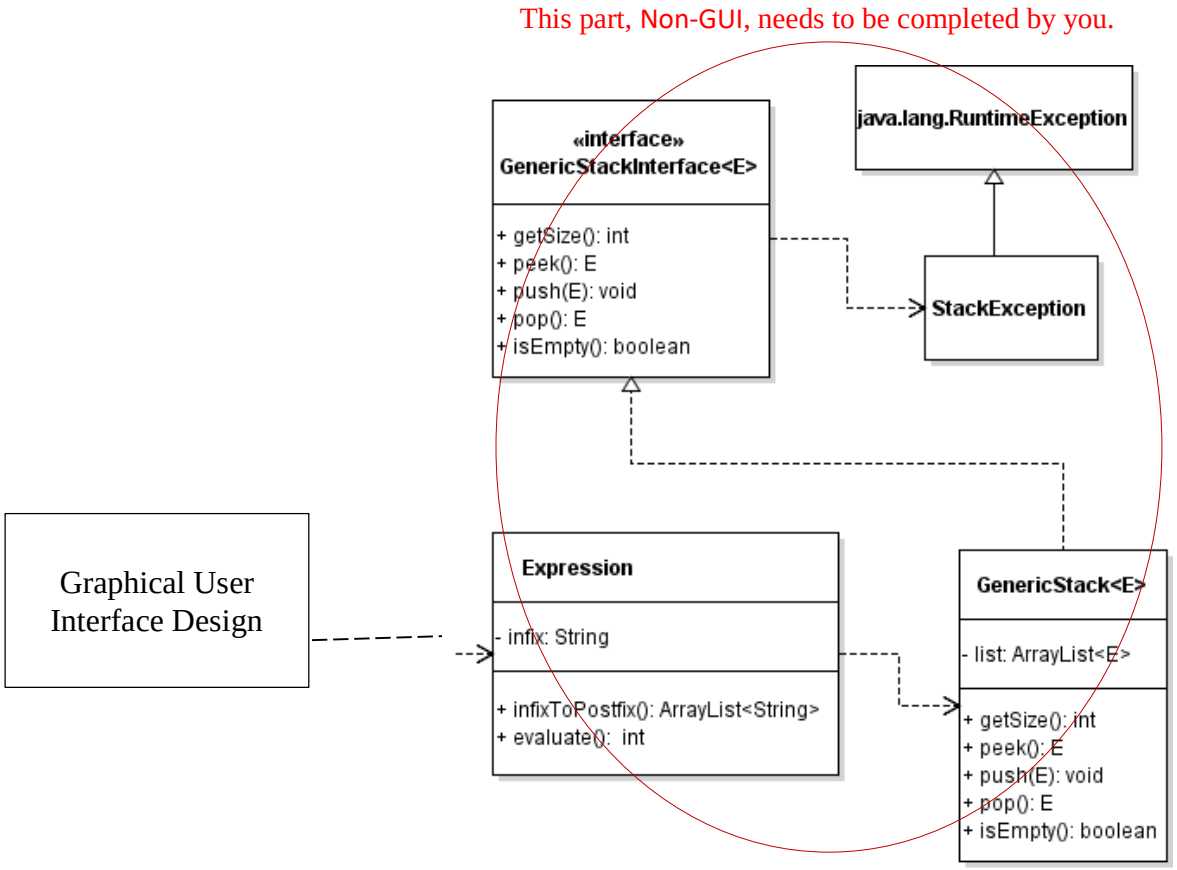
2*((3+5)+4)

2*(3+(5+4))

2*((4+3)+5)

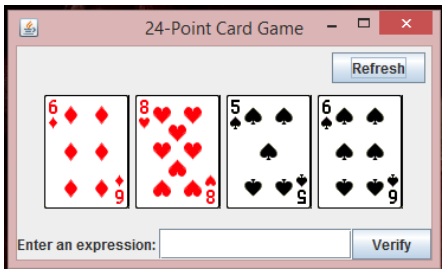
Design:

The following is the design diagram. Part of the design (Graphical User Interface, GUI) has been done. You need to complete the other part of the design and submit only that part of the design when submitting the project.

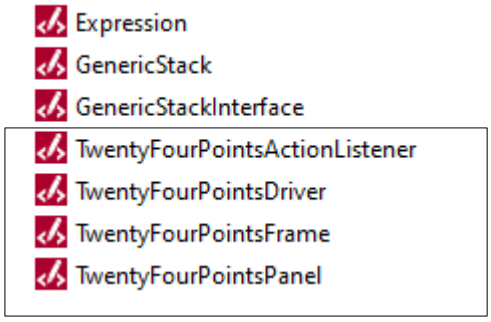


Class *Expression* is instantiated and *evaluate* of the class is invoked in the GUI part. All exceptions must be handled in the Non-GUI design.

- **GUI Design provided**:
  To store the 58 images, you must create folder *image* in the project folder, create a subfolder *card*, and store all images in folder *card*.
  Note: This is required so that we can run your project without errors. You don't need to submit the images as part of the project. The same images will be used for grading.



The four Java programs for GUI have been provided for you to use. You are not allowed to change the GUI programs. You need to complete the other three programs, and test the project completely.



GUI Programs

- **<u>Non-GUI design to be completed by you:</u>(No static methods are allowed.)**

  In this application, an expression(an infix expression) is entered according to the cards shown, and the expression is evaluated. The result is correct if it is 24. To evaluate an infix expression, convert it to an equivalent postfix expression, and evaluate the postfix. It is required to use the infix to postfix algorithm and postfix evaluation algorithm discussed in class for this project.

  In this part of the design, two types of objects need to be designed: ***Expression* and *ADT Stack***. An expression object contains an infix expression (a string) and operations such as *evaluate* and *infixToPostfix*. In both operations, an ADT Stack is needed. Method *evaluate* should convert this infix expression to postfix and evaluate the postfix expression. **It is required to design an ADT Stack and use it in this project. Using *Stack* class from Java library or others will result in zero credit for this project.**

  A Generic ADT Stack:

  Both *infixToPostfix* algorithm and *evaluate* algorithm must use a stack to store tokens. A generic ADT Stack must be designed(*GenericStackInterface*) and implemented using **an array list**. It should have the following operations:

  - Construct an empty stack (*GenericStack(){…}*).
  - Return the number of objects of this stack(*getSize() {… }*).
  - Return a reference to the top element of this stack(*peek() {…}*).
  - Add an object to the top of this stack(*push(E o) {…}*).
  - Remove from the top of this stack(*pop() {…}*).
  - Indicate if this stack is empty(*isEmpty() {…}*).

```
public class GenericStack<E>{
      private java.util.ArrayList<E> list;
    …
}
```

  Expression:

  Define Class *Expression* including the method *infixToPostfix()* and method *evaluate()*. Other methods such as constructors and helper methods should be included in the class. From the GUI side, *Expression* constructor is called with a string passed into the constructor, and an *Expression* object can call the *evaluate* method that returns the result of the expression. See the example below.

```
…
Expression exp = new Expression(A string literal);
.. exp.evaluate() == 24 …
```

…

You can't change the GUI source codes. You are responsible for handling all the exceptions in the Non-GUI part.

infixToPostfix: Converts this infix expression to an equivalent postfix expression, returns a postfix expression as a list of tokens.

```
public ArrayList<String> infixToPostfix(){…}
```

I**nfix to postfix** algorithm (using a stack):

As long as there are more tokens, get the next token.

If the token is an operand, append it to the postfix string.

If the token is "(", push it onto the stack.

If the token is an operator, (order operators by precedence)

if the stack is empty, push the operator onto the stack.

if the stack is not empty, pop operators of greater or equal precedence from the stack and append them to postfix string, stop when you encounter ")" or an operator of lower precedence or when the stack is empty. And then, push the new operator onto the stack.

When you encounter a ")", pop operators off the stack and append them to the end of the postfix string until you encounter matching "(".

When you reach the end of the infix string, append the remaining content of the stack to the postfix String.

evaluate: This method should invoke *infixToPostfix* to convert this infix expression to postfix, evaluate the postfix expression, and return the result.

```
public int evaluate(){…}
```

Helper methods: You may need to write helper methods such as methods for checking precedence in this class

When splitting an infix expression into tokens, you should not use *charAt* method and expect all operands are single-digit tokens. Instead you should split an infix expression by proper delimiters so that all operators, operands (single-digit or not), and parentheses are tokens. One way is to use operators and parentheses as delimiters to split an infix expression and also use them as regular tokens. Please check *StringTokenizer* class or *String* class for proper methods that can be used.

Code/Test/Debug:

Test the entire program by shuffling the cards to make different infix expressions.